1. Tempo disponibile 120 minuti (90 minuti per gli studenti di "Introduzione agli Algoritmi" - 6 CFU, che devono fare solo i primi 3 esercizi).
2. Non è possibile consultare appunti, slide, libri, persone, siti web, ecc.
3. Scrivere in modo leggibile, su ogni foglio, nome, cognome e numero di matricola.
4. Le soluzioni agli esercizi che richiedono di progettare un algoritmo devono:
   - spiegare a parole l'algoritmo (se utile, anche con l'aiuto di esempi o disegni),
   - fornire e commentare lo pseudo-codice (indicando il significato delle variabili),
   - calcolare la complessità (con tutti i passaggi matematici necessari),
   - se l'esercizio ammette più soluzioni, a soluzioni computazionalmente più efficienti e/o concettualmente più semplici sono assegnati punteggi maggiori.

1. Calcolare la complessità $T(n)$ del seguente algoritmo MYSTERY:

Algorithm 1: MYSTERY(INT n) $\rightarrow$ INT

```
if n ≤ 1 then
    return 1
else
    Int v ← 0
    for i ← 1 to n do
        v ← v + i
    end
    return v + Mystery2(n − 1)
end

function Mystery2(Int m) → Int
if m ≤ 1 then
    return 1
else
    Int w ← 0
    for j ← m downto 1 do
        w ← w + j
    end
    return w + Mystery(m − 1)
end
```

Soluzione. Indichiamo con $T(n)$ e $T'(m)$ i costi computazionali delle due invocazioni di funzioni MYSTERY(n) e MYSTERY2(m), rispettivamente. Visto che le due funzioni si richiamano vicendevolmente, abbiamo che:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ c_2 \times n + T'(n-1) & \text{altrimenti} \end{cases} \qquad T'(m) = \begin{cases} c'_1 & \text{se } m \leq 1 \\ c'_2 \times m + T(m-1) & \text{altrimenti} \end{cases}$$

Come primo passaggio, nella prima relazione, sostituiamo $T'(n-1)$ con la relativa definizione, ottenendo:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ c_2 \times 2 + c'_1 & \text{se } n = 2 \\ c_2 \times n + c'_2 \times (n-1) + T(n-2) & \text{altrimenti} \end{cases}$$

Ora abbiamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} c' & \text{se } n \leq 2 \\ c_2 \times n + c_2' \times (n-1) + T(n-2) & \text{altrimenti} \end{cases}$$

che risolviamo tramite iterazione:

$$\begin{aligned} T(n) &= c_2 \times n + c_2' \times (n-1) + T(n-2) \\ &= \Theta(n) + \Theta(n-1) + T(n-2) \\ &= \Theta(n) + \Theta(n-1) + \Theta(n-2) + \Theta(n-3) + T(n-4) \\ &= \ldots = \sum_{i=1}^{n} \Theta(i) = \Theta(\tfrac{n\times(n+1)}{2}) = \Theta(n^2) \end{aligned}$$

2. Scrivere un algoritmo che, preso in input un albero binario T ed un intero positivo k, ritorni il numero di foglie a profondità maggiore o uguale di k. Discutere la complessità nel caso pessimo e nel caso ottimo. Nota: la radice si trova a profondità 0.

   Soluzione. Implementiamo una funzione ricorsiva (ricerca in profondità) che decrementa il parametro k ad ogni chiamata ricorsiva. Il paramento k viene decrementato solo se è maggiore di zero (ricordiamo che la funzione deve prendere in input un intero positivo). Abbiamo due casi base: 1) il primo gestisce il caso in cui l'albero in input sia vuoto. In questo caso la funzione ritorna 0. 2) il secondo gestisce il caso in cui il nodo corrente sia una foglia e il parametro $k = 0$ (questo implica che abbiamo raggiunto/superato la profondità richiesta nella prima chiamata a funzione). In questo caso la funzione ritorna 1.

Algorithm 2: CountLeaves(BinTree T, Int k) → Int

```
if T = NULL then
 | return 0
else if T.isLeaf() and k = 0 then
 | return 1
else
 | return CountLeaves(T.left,Max(0,k-1))+CountLeaves(T.right,Max(0,k-1))
```

   Sia nel caso pessimo che nel caso ottimo siamo costretti a visitare tutti i nodi dell'albero per poter individuare le foglie a profondità maggiore o uguale a k. Il costo computazionale dell'algoritmo à quindi $\Theta(n)$, dove n è il numero di nodi nell'albero T.

3. Un ladro entra in un appartamento in cui si trovano n oggetti, ognuno con un proprio valore $v[i]$. Ha a disposizione k borse da riempire con gli oggetti da rubare, con il vincolo che **ogni borsa** può contenere **un solo** oggetto. Bisogna aiutare il ladro a calcolare il massimo valore possibile del bottino. Dovete quindi progettare un algoritmo che dati i valori n (numero oggetti), k (numero borse), e l'array $v[1..n]$ (valori degli oggetti), restituisce il valore complessivo massimo che è possibile ottenere selezionando k oggetti. Riportare il costo computazionale (in tempo) della soluzione proposta, nel caso pessimo, nel caso ottimo, e possibilmente anche nel caso medio.

   Soluzione. Si può utilizzare un approccio greedy che trova il massimo bottino semplicemente sommando i k oggetti di maggiore valore. Un modo efficiente, nel caso medio, per selezionare i k oggetti di maggiore valore utilizza una versione di quickselect per cercare il $(n-k)$-esimo minimo utilizzando sempre lo stesso array, partizionandolo utilizzando il cosiddetto algoritmo della bandiera nazionale discusso a lezione; al termine della ricerca dell'$(n-k)$-esimo minimo, tale algoritmo colloca nelle ultime k posizioni dell'array i k elementi di maggiore valore. Dopo aver eseguito questa versione di quickselect per cercare l'$(n-k)$-esimo minimo, sarà sufficiente sommare il valore degli elementi che si trovano nelle ultime k posizioni.

   L'algoritmo SommaMassimi (si veda Algoritmo 3) è una versione iterativa di quickselect che ha quindi il medesimo costo computazionale studiato a lezione, in quanto l'aggiunta del ciclo finale (di costo $\Theta(k)$) non cambia la classe di complessità, che risulta quindi essere lineare nel caso ottimo e medio, quadratica nel caso pessimo.

Cont.

Algorithm 3: SommaMassimi(Int n, Int k, Number $v[1 \dots n]$) → Number

```
s ← 1; e ← end     // s ed e delimitano il sottovettore su cui si sta effettuando la ricerca
while true do
    scegli casualmente un valore v[w] con w compreso tra s e e
    (iniziox, finex) ← Partiziona(v[1...n], s, e, v[w])
    if n − k < iniziox then
        e ← iniziox − 1                  // l'elemento scelto è maggiore dell'(n − k)-esimo minimo
    else if n − k > finex then
        s ← finex + 1                    // l'elemento scelto è minore dell'(n − k)-esimo minimo
    else
        exit                             // l'elemento scelto è il (n − k)-esimo minimo
    end
end
/* restuituice la sommatoria dei valori nelle ultime k posizioni di v                    */
somma ← 0
for i ← n − k + 1 to n do
    somma ← somma + v[i]
end
return somma
```

Algorithm Partiziona(Int $v[1 \dots n]$, Int *start*, Int *end*, Number x) → (Int, Int)

```
/* bandiera nazionale:  minori di x verdi, uguali a x bianchi, maggiori di x rossi     */
/* restituisce una coppia di indici:  inizio e fine della collocazione dei valori x    */
i ← start; j ← start; k ← end
while j < k do
    if A[j] < x then
        if i < j then
            scambia A[i] con A[j]
        end
        i ← i + 1; j ← j + 1
    else if A[j] > x then
        scambia A[j] con A[k]
        k ← k − 1
    else
        j ← j + 1                                          // in questo caso A[j] = x
    end
end
return (i, j − 1)          // i valori x (bianchi) sono dalla posizione i alla posizione j − 1
```

4. Dato un grafo non orientato connesso $G = (V, E)$ ed un suo vertice $v \in V$, definiamo **raggio** del vertice v in G, la massima distanza fra v ed un qualsiasi altro vertice in V (si ricorda che la distanza fra due vertici è il numero minimo di archi di un cammino fra tali vertici). Matematicamente:

$$raggio(v, G) = max\{u \in V \mid distanza(v, u)\}$$

Progettare un algoritmo che dato un grafo non orientato connesso $G = (V, E)$ ed un vertice $v \in V$, restituisce $raggio(v, G)$.

Soluzione. È sufficiente effettuare una visita in ampiezza del grafo a partire da v e restituire la distanza dell'ultimo nodo visitato, visto che la BFS visita i vertici in ordine non decrescente di distanza dal vertice di inizio della visita. L'algoritmo (si veda Algoritmo 4) ha la medesima complessità della visita in ampiezza che, assumendo implementazione tramite liste di adiacenza, e sapendo che il grafo è connesso (quindi con un numero di archi maggiore o uguale, in ordine di grandezza, del numero di vertici), risulta essere $\Theta(|E|)$, in quanto tutti gli archi vengono controllati almeno una volta.

Cont.

Algorithm 4: Raggio(Graph $G = (V, E)$, Vertex v) $\rightarrow$ Int

```
Queue q ← new Queue()
for x ∈ V do
    x.mark ← false
    x.dist ← ∞
end
v.mark ← true
v.dist ← 0
q.enqueue(v)
while not q.isEmpty() do
    u ← q.dequeue()
    for w ∈ u.adjacents() do
        if not w.mark then
            w.mark ← true
            w.dist ← u.dist + 1
            q.enqueue(w)
        end
    end
end
return u.dist                                   // u contiene l'ultimo vertice visitato
```

The End.